Patent Application of
Kevin W Jameson
For

COLLECTION MAKEFILE GENERATOR

CROSS REFERENCES TO RELATED APPLICATIONS

The present invention uses inventions from the following patent applications that are filed contemporaneously herewith, and which are incorporated herein by reference:

USPTO 09/885078, Collection Information Manager; Kevin Jameson.
USPTO 09/885076, Collection Content Classifier; Kevin Jameson.

FIELD OF THE INVENTION

This invention relates to automated software systems for processing collections of computer files in arbitrary ways, thereby improving the productivity of software developers, web media developers, and other humans and computer systems that work with collections of computer files.

important disadvantages. Notably, general prior art mechanisms do not provide fully automated support for collections, dynamic determination of content files, multiple products, extensive makefile variance, or parallel execution support.

In contrast, the present collection makefile generator invention has none of these limitations, as the following disclosure will show.

Specific Shortcomings in Prior Art

Several examples of prior art makefile generators are discussed below. The examples fall into two main categories: makefile generator programs and integrated development environment (IDE) programs. Both types of programs generate makefiles so that project source files can be processed efficiently in an automated manner.

Prior Art Makefile Generators

Makefile generator programs generate makefiles for humans who are building software programs. Typically, makefiles contain computer instructions for compiling source code files and linking compiled object files to produce executable files or libraries of object files. Also typically, programmers include a variety of other useful command sequences in makefiles to increase productivity.

Some examples of popular freeware makefile generators include automake, imake, and mkmf (make makefile). One example of a patented makefile generator is US Patent 5872977 "Object-Oriented Method and Apparatus For Creating A Makefile" by Thompson, which describes an object-oriented method of generating makefiles from input build files and input rule files. Although each of these prior art approaches is useful in some way, each approach has several important shortcomings.

Automake has no dynamic content discovery mechanism; instead it requires programmers to manually list all files that require processing. Neither does it have a

151 Calculate include search directories

152 Do file type definition services module

153 Do action type definition services module

160 Process makefile service module

161 Substitute makefile fragment module

162 Insert makefile fragment module

DETAILED DESCRIPTION

Overview of Collections

This section introduces collections and some related terminology.

Collections are sets of computer files that can be manipulated as a set, rather than as individual files. Collection information is comprised of three major parts: (1) a collection specifier that contains information about a collection instance, (2) a collection type definition that contains information about how to process all collections of a particular type, and (3) optional collection content in the form of arbitrary computer files that belong to a collection.

Collection specifiers contain information about a collection instance. For example, collection specifiers may define such things as the collection type, a text summary description of the collection, collection content members, derivable output products, collection processing information such as process parallelism limits, special collection processing steps, and program option overrides for programs that manipulate collections. Collection specifiers are typically implemented as simple key-value pairs in text files or database tables.

FIG 1 shows an example prior art filesystem folder from a typical personal computer filesystem. The files and directories shown in this drawing do not implement a collection 100, because no collection specifier 102, FIG 2 Line 5 exists to associate a collection type definition FIG 4 101 with collection content information FIG 4 103.

FIG 2 shows the prior art folder of FIG 1, but with a portion of the folder converted into a collection 100 by the addition of a collection specifier file FIG 2 Line 5 named "cspec". In this example, the collection contents FIG 4 103 of collection 100 are defined by two implicit policies of a preferred implementation.

First is a policy to specify that the root directory of a collection is a directory that contains a collection specifier file. In this example, the root directory of a collection 100 is a directory named "c-myhomepage" FIG 2 Line 4, which in turn contains a collection specifier file 102 named "cspec" FIG 2 Line 5.

Second is a policy to specify that all files and directories in and below the root directory of a collection are part of the collection content. Therefore directory "s" FIG 2 Line 6, file "homepage.html" FIG 2 Line 7, and file "myphoto.jpg" FIG 2 Line 8 are part of collection content FIG 4 103 for said collection 100.

FIG 3 shows an example physical representation of a collection specifier file 102, FIG 2 Line 5, such as would be used on a typical personal computer filesystem.

Collection Information Types

Figures 4-5 show three kinds of information that comprise collection information.

FIG 4 shows a high-level logical structure of three types of information that comprise collection information: collection processing information 101, collection specifier information 102, and collection content information 103. A logical collection 100 is comprised of a

collection specifier 102 and collection content 103 together. This diagram best illustrates the logical collection information relationships that exist within a preferred filesystem implementation of collections.

FIG 5 shows a more detailed logical structure of the same three types of information shown in FIG 4. Collection type definition information FIG 4 101 has been labeled as per-type information in FIG 5 103 because there is only one instance of collection type information 101 per collection type. Collection content information FIG 4 103 has been labeled as per-instance information in FIG 5 103 because there is only one instance of collection content information per collection instance. Collection specifier information 102 has been partitioned into collection instance processing information 104, collection-type link information 105, and collection content link information 106. FIG 5 is intended to show several important types of information 104-106 that are contained within collection specifiers 102.

Suppose that an application program means FIG 6 110 knows (a) how to obtain collection processing information 101, (b) how to obtain collection content information 103, and (c) how to relate the two with per-collection-instance information 102. It follows that application program means FIG 6 110 would have sufficient knowledge to use collection processing information 101 to process said collection content 103 in useful ways.

Collection specifiers 102 are useful because they enable all per-instance, non-collection-content information to be stored in one physical location. Collection content 103 is not included in collection specifiers because collection content 103 is often large and dispersed among many files.

All per-collection-instance information, including both collection specifier 102 and collection content 103, can be grouped into a single logical collection 100 for illustrative purposes.

Virtual Platforms

As can be appreciated from the foregoing discussion, a large number of makefile fragments are required to effectively model the makefile needs of a typical industrial software environment. For example, several hundreds of fragments might be involved.

One helpful technique for managing large numbers of fragments is to organize them into virtual platform directories, and then use virtual platform search directories to find specific fragment files. A virtual platform is one that is invented by fragment administrators to represent a desired abstraction level for sharing makefile information.

There are two main benefits of this approach.

The first benefit is that information can be more easily shared at various operating system abstraction levels. For example, virtual platform "pi" information can be shared among all platforms, virtual platform "gnulinux" information can be shared among all gnulinux systems, and virtual platform "gnulinux2" information can be used only by gnulinux2 systems.

The second benefit is that virtual platform search rules make it possible to more easily override more generic information with more specific information. For example, placing "gnulinux2" ahead of "pi" in a set of virtual platform search rules ensures that the "gnulinux2" version of a same-named file will always be found before the "pi" version of the same-named file.

FIG 59 shows a table of virtual platforms, with associated search directories.

FIG 60 shows two examples of how virtual platform entries from the table of FIG 59 can be converted into virtual platform search rules, or search directories.

For example, FIG 59 Line 6 contains a virtual platform entry for the "gnulinux2" virtual

Administrative parallelism is determined by administrative policy. This is because system administrators may want to limit the computational resources that can be accessed by any one parallel computation. For example, parallel calculations can generate significant amounts of computer load, so system administrators may want to protect other system users from big parallel calculations that hog scarce computational resources.

Useful parallelism is the maximum amount of parallelism that can usefully be applied to a particular computation under particular parallelism limits. Suppose that administrative parallelism limits are set high enough to be ignored. Then useful parallelism would be calculated as the minimum of problem parallelism and physical parallelism.

Parallel Target Examples

FIG 62 shows examples of parallel targets and how they are used.

Line 3 shows a sequential makefile target that has 100 object file dependencies. As written, Line 3 will cause the 100 object files to be compiled in sequence.

Line 8 shows how parallelism can be obtained using a make program which is capable of spawning multiple parallel computational processes using sequential targets. Make accepts a control argument "-j N" that specifies the number of parallel processes to spawn. In this example, 4 parallel processes are requested. Thus make would simultaneously build file-001.o through file-004.o.

Lines 12-16 shows an example set of parallel targets that might be generated by the present makefile generator invention. Line 12 is the top level target, with 4 dependencies that are the 4 parallel targets.

Lines 13-16 are parallel targets, each responsible for building 25 of the 100 example object files.